Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2013, Article ID 147593, 12 pages
http://dx.doi.org/10.1155/2013/147593

Research Article

An Abstract Description Method of Map-Reduce-Merge Using Haskell

Lei Liu,[1] Dongqing Liu,[1] Shuai Lü,[1,2] and Peng Zhang[1]

[1] *College of Computer Science and Technology, Jilin University, Changchun 130012, China*
[2] *College of Mathematics, Jilin University, Changchun 130012, China*

Correspondence should be addressed to Shuai Lü; lus@jlu.edu.cn

Received 17 May 2013; Accepted 17 July 2013

Academic Editor: William Guo

Map-Reduce-Merge is an improved parallel programming model based on Map-Reduce in cloud computing environment. Through the new Merge module, Map-Reduce-Merge can support processing multiple related heterogeneous datasets more efficiently. In order to demonstrate the validity and effectiveness of this new model, we present a rigorous description for Map-Reduce-Merge model using Haskell. Firstly, we describe the basic program skeleton of Map-Reduce-Merge programming model. Secondly, an abstract description for the Merge module is presented by analyzing the structure and function of the Merge module with Haskell as the description tool. Thirdly, we evaluate the Map-Reduce-Merge model on the basis of our description. We capture the functional characteristics of the Map-Reduce-Merge model by our abstract description, which can provide theoretical basis for designing more efficient parallel programming model to process join operation.

1. Introduction

Recently, lots of research works on improving Google's Map-Reduce [1] model have been proposed to analyze large volumes of data [2–5]. One important type of data analysis is joining multiple datasets. There are increasing efforts for implementing join algorithms using Map-Reduce or the improved Map-Reduce programming model [6–11]. Map-Reduce-Merge is such an effort that can directly express join operation and implement several join algorithms by the new Merge module. In this new model, Map and Reduce modules are inherited from Map-Reduce model, so that existing Map-Reduce programs can run directly on this new framework without modifications. Not only join operator but also all the other relational operators can be modeled using various combinations of the three primitives: Map, Reduce, and Merge. Map-Reduce-Merge removes the burden of implementing join algorithms. The emergency of Map-Reduce-Merge shows a trend that parallel databases and Map-Reduce learn with each other and new data analysis ecosystems are developed [12, 13].

Many formal methods can be used to describe programming model [14–17]. Lämmel [18] first delivers a rigorous description of Map-Reduce programming model as well as its advancement called Sawzall [19]. He uses typed functional programming Haskell as a tool to describe the fundamental characteristics underlying the Map-Reduce and Sawzall model. The description is made up of several Haskell functions. Our paper is based on his work. We will present an abstract description for Map-Reduce-Merge model, especially for the new added Merge module. This paper makes the following contributions. Firstly, we define the basic program skeleton of Map-Reduce-Merge to capture the abstraction for Map-Reduce-Merge computations. Secondly, we decompose the Merge module according to its structure and present the rigorous description called *moduleMerge*. Thirdly, we analyze the Map-Reduce-Merge programming model based on our abstract description with an example. Some implementation details (such as fault tolerance and task scheduling [20–25]) will be considered in the future.

Haskell is characterized by strong type inference and type checking [26]. The recent paper [18] suggests that Haskell can be used as a tool to support executable specification. Using Haskell as a description tool can be beneficial for both programming model designers and users. For designers, they can know explicitly what will happen during the execution of a Map-Reduce-Merge job, which is good for them to analyze

```
map :: (a → b) → [a] → [b ]              --type of map
map f [] = []                            --the empty list case
map f (x: xs) = f x : map f xs           --the non-empty list case
```

Algorithm 1

```
map (+1) [1,2,3]
= 2: map (+1) [2,3]
= 2: 3: map (+1)[3]
= 2: 3: 4: []
= [2, 3, 4]
```

Algorithm 2

and evaluate this new model. For users, they can use the abstract description as an executable specification in software development to ensure the correctness and robustness of software.

This paper is organized as follows. Section 2 gives a brief introduction of Map-Reduce-Merge programming model and Haskell programming language. Section 3 defines the basic programming skeleton of Map-Reduce-Merge programming model. Section 4 designs the helper functions composing the Merge module. Section 5 defines the helper functions designed in Section 4. Section 6 shows an example and gives a brief comment. Section 7 concludes this paper.

2. Background

In this section, we will briefly recall Map-Reduce-Merge programming model and some concepts in Haskell.

2.1. Map-Reduce-Merge Programming Model. The most important feature of Map-Reduce-Merge is that it adds to Map-Reduce a Merge phase, so that it can directly support join algorithms of different datasets. Figure 1 illustrates the data flow of the Map-Reduce-Merge framework. It consists of three phases, two independent Map-Reduce phases, and a Merge phase. At first, two datasets are processed by corresponding Map and Reduce phases. Then in the Merge phase, a merger can select data to be merged from those two reducer outputs according to different join algorithm. Notice that the reducer outputs are stored in local disks instead of distributed file system because the Reduce phase is not the last phase any more.

The main purpose of the Merge module is to implement join algorithms. The Merge module includes four components: *Partition Selector*, *Processors*, *Configurable Iterators*, and *Merger* as shown in Figure 2. *Partition Selector* is a selector that determines which data partitions produced by up-stream reducers should be retrieved and then merged. *Configurable Iterators* include two logical iterators which can implement different join algorithms, including sort-merge join, nested-loop join, and hash join. *Processors* include two processor functions which define the logic to process data from different datasets. *Merger* includes a merger function where users can implement data processing logic on data from two sources.

2.2. Haskell Programming Language. To avoid confusion, we will introduce the *map* function and the *Map* type in Haskell, as well as the map primitive in Map-Reduce programming model.

In Haskell, the *map* function is a higher-order function. It takes two parameters, the first is a function whose type is $a \rightarrow b$, and the second is a list whose type is $[a]$. It returns a list whose type is $[b]$. The formal definition of *map* is as in Algorithm 1.

We illustrate how to use *map* to add one to every element in a list. Here, the expression (+1) represents a function that adds one to a variable, which equals to the function $f(x) = x + 1$. The example is shown as in Algorithm 2.

The type *Map* is a build-in type in Haskell. It is an efficient implementation of maps from keys to values. We can use the function *toList* to convert a *Map* to an association list and the function *fromList* to build a *Map* from an association list.

The signature of map primitive is $(K1, V1) \rightarrow [(K2, V2)]$. The map primitive in essence corresponds to the first parameter of the function *map* in Haskell [18]. We define the signatures of map, reduce, and merge primitives in Table 1. The first digit after letter K/V is used to distinguish different keys/values, while the second digit is used to distinguish different datasets.

3. Definition of *mapReduceMerge*

In this section, we define the *mapReduceMerge* function to model the abstraction for Map-Reduce-Merge computations. A full Map-Reduce-Merge job includes two individual Map-Reduce phases and a Merge phase as shown in Figure 1. Hence, we take for granted that the *mapReduceMerge* function can be decomposed into three helper functions that represent these three phases, respectively. The type and definition for *mapReduceMerge* are shown as in Algorithm 3.

The *mapReduceMerge* function is defined in terms of function application in Haskell. The arguments *lTable* and *rTable* corresponding to the types $[Map\ K11\ V11]$ and $[Map\ K12\ V12]$ are the input data for a Map-Reduce-Merge job. They are first processed by the functions *mapReduce*1 and *mapReduce*2, respectively, and then are merged by the function moduleMerge. These helper functions correspond to three phases in a Map-Reduce-Merge computation.

By taking advantage of the works done in [18], a Map-Reduce job is divided into three phases: Map, Shuffle, and Reduce. We can define *mapReduce* as in Algorithm 4.

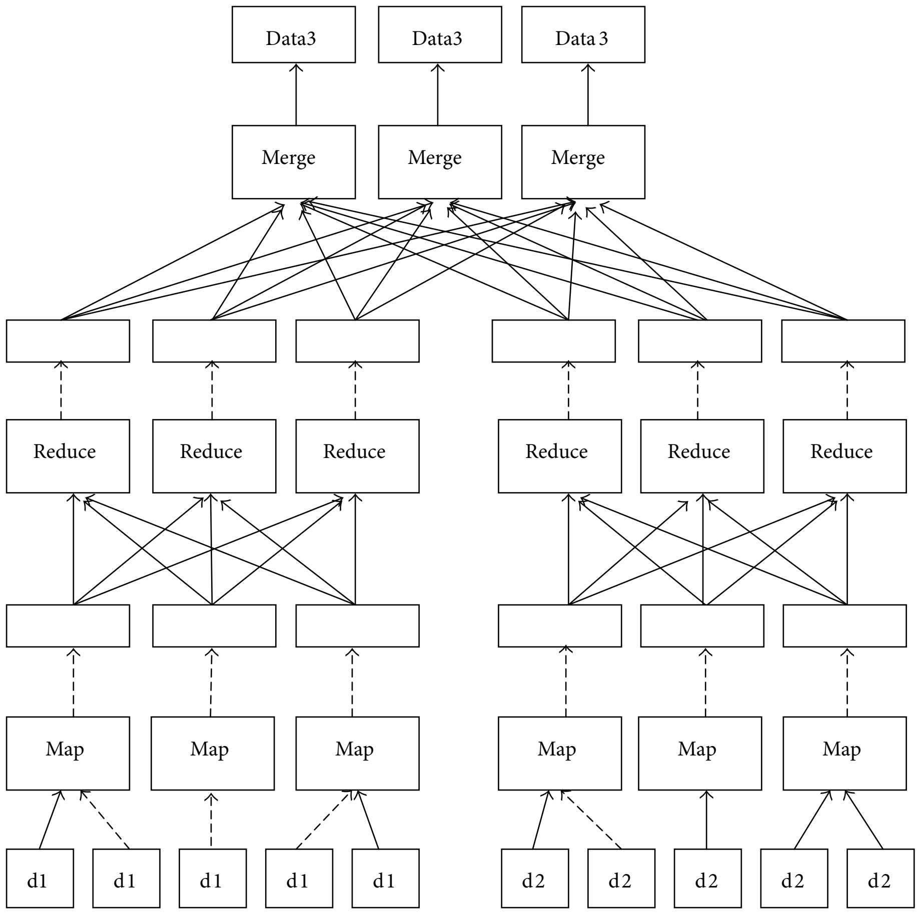

FIGURE 1: Data flow for the Map-Reduce-Merge framework.

Here, function composition, which is denoted by Haskell infix operator (.), is used to compose three helper functions. A more detailed definition for *mapReduce* can be found in [18]. In this paper, we mainly focus on describing the Merge module with the *moduleMerge* function. By analyzing *mapReduceMerge* and *mapReduce*, we can discover the type for *moduleMerge* with its definition to be defined later (Algorithm 5).

Now, we will explain the types used in the definition of *mapReduceMerge*. The type *Map K*1 *V*1 represents the input split type for *moduleMap*. Typically, every split corresponds to a map task. Hence, the list type [*Map K*1 *V*1] represents all the input data of a Map-Reduce job. Similarly, the type *Map K*2 *V*3 represents the output result of a reduce task, and the list type [*Map K2 V3*] represents all the output data of a Map-Reduce job. Our types are compatible with the types defined in [18].

In parallel databases, a table is divided into different splits in order to store in large clusters. The splits form the unit of distribution and load balancing. In this paper, we use *Table* to represent a table, *Tablet* to represent a split of a table, and *Record* to represent a row in a table. According to our discussion above, *Table* has a type of [*Map K V*], *Tablet* has a type of *Map K V*, and *Record* has a type of (*K, V*). It happens that a table in Google's Bigtable [27] is a sparse, distributed, persistent multidimensional sorted *Map*.

4. Discovery of the Types in Merge

In this section, we design some helper functions for *moduleMerge* and discover the types of those functions. The *moduleMerge* function is defined to model the abstraction for the Merge module. According to Figure 2, the components of the Merge module can be divided into two parts, data transferring and data processing. First part includes *Partition Selector*, which can select and transfer the output of up-stream reducers. Second part is made up of *Processors*, *Configurable Iterators*, and *Merger*, which can merge two different datasets. Hence, we design two helper functions for *moduleMerge*, which are *getPartitionPair* and *mergeTwoPartition*. The *getPartitionPair* function selects the output *Tablets* from up-stream reducers and then delivers those Tablets to mergers. The *mergeTwoPartition* function takes two Tablets as input and returns a new *Tablet* as the merge result. The types of these two functions are shown as in Algorithm 6.

Here, [*Map K21 V31*] denotes the input type of *moduleMerge*. It coincides with the result type of *mapReduce*1, which reflects the fact that the output of *mapReduce*1 is one input for *moduleMerge*. The type *(Map K*21 *V*31*, Map K*22 *V*32*)* represents two *Tablets* to be merged. The type [(*Map K*21 *V*31, *Map K*22 *V*32)] represents all the *Tablet* pairs that is a merger, process and the type [[(*Map K*21 *V*31, *Map K*22 *V*32)]] represents all the *Tablet* pairs that is all the mergers process.

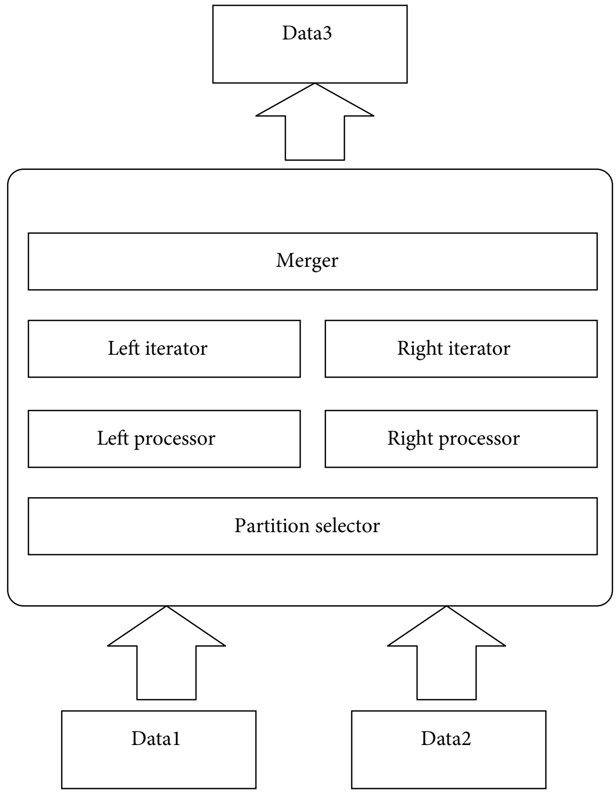

FIGURE 2: Data flow for the Merge module.

TABLE 1: The signatures of three Map-Reduce-Merge primitives.

	Source 1	Source 2
Map	$K11 \rightarrow V11 \rightarrow [(K21, V21)]$	$K12 \rightarrow V12 \rightarrow [(K22, V22)]$
Reduce	$K21 \rightarrow [V21] \rightarrow Maybe\ V31$	$K22 \rightarrow [V22] \rightarrow Maybe\ V32$
Merge	$(K21, V31) \rightarrow (K22, V32) \rightarrow (K23, V33)$	

```
mapReduceMerge :: [Map K11 V11] → [Map K12 V12] → [Map K23 V33]
mapReduceMerge lTable rTable = moduleMerge (mapReduce1 lTable)(mapReduce2 rTable)
```

ALGORITHM 3

```
mapReduce :: [Map K1 V1] → [Map K2 V3]
mapReduce = moduleReduce.moduleShuffle.moduleMap
```

ALGORITHM 4

```
moduleMerge :: [Map K21 V31] → [Map K22 V32] → [Map K23 V33]
moduleMerge = undefined
```

ALGORITHM 5

```
getPartitionPair :: [Map K21 V31]                    --Table1 as distributed input data
                 → [Map K22 V32]                     --Table2 as distributed input data
                 → [[(Map K21 V31, Map K22 V32)]]    --The tasks of all mergers
mergeTwoPartition :: Map K21 V31       --Tablet1 as input data
                  → Map K22 V32        --Tablet2 as input data
                  → Map K23 V33        --Tablet3 as output data
```

Algorithm 6

```
getPair :: Map K21 V31          --Tablet1 as input data
        → Map K22 V32           --Tablet2 as input data
        → [(KV1, KV2)]          --The task of one merger
mergeTwoRecord :: KV1                  --Record1 as input data
               → KV2                   --Record2 as input data
               → Maybe KV3             --Maybe Record3 or Nothing as output data
```

Algorithm 7

Every merger has a serial number in the Map-Reduce-Merge programming model. We implicitly express this characteristic by the merger position in the list.

We decompose the *mergeTwoPartition* function into two helper functions called *getPair* and *mergeTwoRecord*. The *getPair* function chooses and emits all the *Record* pairs that will be processed by the *mergeTwoRecord* function, where a *Record* pair is merged into a new *Record*. In fact, *mergeTwoPartition* is implemented by executing *mergeTwoRecord* many times. The types of these two functions are shown as in Algorithm 7.

Here, the type $KV1$ is short for the type $(K21, V31)$, $KV2$ is short for $(K22, V32)$, and $KV3$ is short for $(K23, V33)$. The type $[(KV1, KV2)]$ represents all the possible *Record* pairs that need to be merged. The type *Maybe* is a built-in type in Haskell, which is defined as "*data Maybe a = Just a | Nothing*." Only those *Record* pairs that satisfy the merge condition can be merged into a new Record whose type is *Just KV3*. The type (*Map K*21 *V*31, *Int*) represents a *Tablet* and its number, which corresponds to the reducer number in Map-Reduce-Merge.

The *getPair* function has to decide whether two *Records* need to be merged or not, so the *iterationLogic* function is designed to implement this work. The same thing happened in the *getPartitionPair* function, where *partitionSelector* is designed to judge whether two *Tablets* need to be merged. The types for *iterationLogic* and *partitionSelector* are shown as in Algorithm 8.

We now summarize the components of the Merge module and the designed functions, as well as the relationship between them. In the Merge module, *Partition Selector* selects the input data for mergers from reducers. We design the *partitionSelector* function to model it. The *partitionSelector* function is invoked by *getPartitionPair* to transfer data from reducers to mergers. Hence, this phase is similar to the shuffle phase in Map-Reduce. *Configurable Iterator* includes two iterators corresponding to two datasets. All join algorithms, such as sort-merge join, nested-loop join, and hash join, have their own processing logic to control the relative movements of two iterators. In our paper, we abstract it by the *iterationLogic* function according to the characteristics of Haskell. The *iterationLogic* function is called by *getPair* to generate all the wanted *Record* pairs. *Merger* that implements the user-defined logic is captured by the *mergeTwoRecord* function. *Processors* include two functions called *leftProcessor* and *rightProcessor*, which can implement hash join. For simplicity, we model it with the *iterationLogic* function. The relationship of our functions is summarized in Figure 3.

5. Definition of the Helper Functions in Merge

In this section, we describe how to implement the functions whose types have been determined in last section. After every function, some concepts of Haskell are explained when needed.

5.1. mergeTwoRecord. The *mergeTwoRecord* function is composed of two helper functions *match* and *mergeResult*. We use the *match* function to judge whether the join keys of two Records satisfy the given condition. If $K21$ and $K22$ satisfy a merge condition defined by the *match* function, these two *Records* will be processed by *mergeResult* where users can implement data processing logic. The definitions of *match* and *mergeResult* are related to concrete applications. In Section 6, we will illustrate how to use these two functions. The definition of *mergeTwoRecord* is as in Algorithm 9.

In the definition of *mergeTwoRecord*, the symbol "@" which is called as-pattern is a kind of pattern matching forms in Haskell. It denotes that $KV1$ can be replaced by $(K21, V31)$ and $KV2$ can be replaced by $(K22, V32)$. The symbol "\$" denotes function application which can be substituted by parentheses. In this definition, "*Just \$ mergeResult KV*1 *KV*2" is equal to "*Just (mergeResult KV*1 *KV*2*).*" This representation is much simpler than using parentheses, especially when nested parentheses are needed. The vertical pipe " " represents guards in Haskell. It is used to judge whether the parameters satisfy some conditions. In this definition, if $K21$ and $K22$

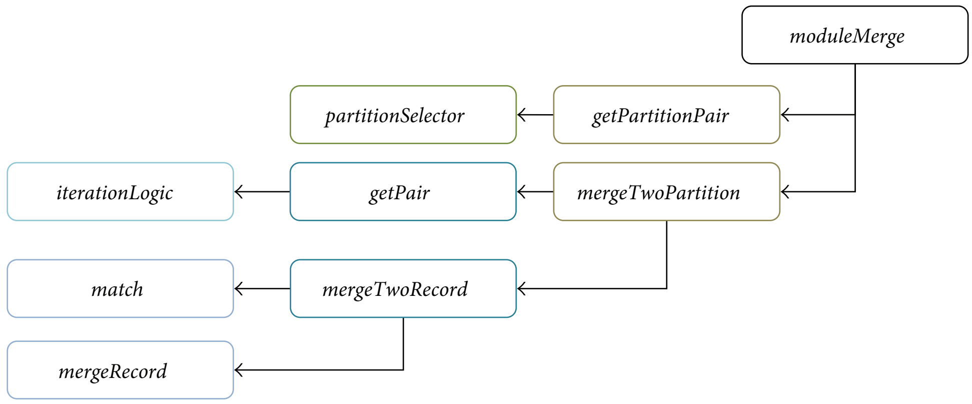

Figure 3: The relationship of the functions in Merge.

```
iterationLogic :: KV1                --Record1 as input data
               → KV2                 --Record2 as input data
               → Bool                --Need to be merged or not
partitionSelector :: (Map K21 V31, Int)     --Tuple (Tablet1, Id1) as input data
                  → (Map K22 V32, Int)      --Tuple (Tablet2, Id2) as input data
                  → Bool                    --Choose Tablet pair to be merged
```

Algorithm 8

```
mergeTwoRecord KV1@(K21, V31) KV2@(K22, V32)
    | match K21 K22 = Just $ mergeResult KV1 KV2
    | otherwise = Nothing
```

Algorithm 9

```
getPair lTablet rTablet =
   filter (uncurry iterationLogic)                                            -- Step 2
   [(pairLeft, pairRight) | pairLeft ← toList lTablet, pairRight ← toList rTablet ]   -- Step 1
```

Algorithm 10

satisfy the condition defined in the *match* function, $KV1$ and $KV2$ will be processed by the *mergeResult* function and then wrapped by the type *Maybe*. Otherwise, *mergeTwoRecord* will return *Nothing* as the result.

5.2. getPair. We implement *getPair* in two steps as follows.

Step 1. Get all possible combinations of *Records* from *lTablet* and *rTablet*; the result is the Cartesian product of these two *Tablets*.

Step 2. Filter the undesired *Record* pairs with *iterationLogic*.

In Haskell, list comprehensions are a handy way to produce lists. Their concepts are similar to set comprehensions in mathematics. Here in Step 1, the list comprehension is used for building a list out of two lists. We could also do the same thing using set comprehensions, like this: $\{(pairLeft, pairRight) \mid pairLeft \in toList\,lTablet, pairRight \in toList\,rTablet\}$. The *filter* function takes a predicate and a list and returns the list whose elements satisfy that predicate. In Step 2, those elements which satisfy the conditions defined in *iterationLogic* will remain in the list, while others will be removed from the list. The definition of *getPair* is as in Algorithm 10.

5.3. mergeTwoPartition. The *mergeTwoPartition* function is implemented in the following four steps.

Step 1. Get the list of pairs from *lTablet* and *rTablet* by invoking *getPair*.

Step 2. Process every pair from the list with *mergeTwoRecord*. As a result; the return type is [*Maybe* $KV3$].

```
MergeTwoPartition lTablet rTablet =
fromList                                                  -- Step 4
$ map (\(Just x) → x) $ filter (\x → x/=Nothing)          -- Step 3
$ map (uncurry mergeTwoRecord)                            -- Step 2
$ getPair lTablet rTablet                                 -- Step 1
```

Algorithm 11

```
getPartitionPair lTable rTable =
map (map (\((lTablet, lNum), (rTablet, rNum)) → (lTablet, rTablet)))       -- Step 4
$ map (filter (uncurry partitionSelcetor))                                 -- Step 3
$ [[(pairLeft, pairRight) | pairRight ← rTableNum] | pairLeft ← lTableNum]  -- Step 2
where                                                                      -- Step 1
      lTableNum = zip lTable [1,2.. ]
      rTableNum = zip rTable [1,2.. ]
```

Algorithm 12

```
moduleMerge lTable rTable =
      map (foldl union empty)                          -- Step 3
      $ map (map (uncurry mergeTwoPartition))          -- Step 2
      $ getPartitionPair lTable rTable                 -- Step 1
```

Algorithm 13

Step 3. Remove the element equal to *Nothing* from the list with the function *filter*. Then change the element type from *Maybe KV*3 to *KV*3 with the map function.

Step 4. Change the list type to the type *Map* with the *fromList* function.

In the definition of the *mergeTwoPartition* function, the symbol "\" denotes λ expression. For example, the λ expression "\x → x/=Nothing" represents a function whose parameter is x and function body is *x/=Nothing*. Hence, this function decides whether x is equal to *Nothing*. The definition of *mergeTwoPartition* is as in Algorithm 11.

5.4. *getPartitionPair*.

We implement the *getPartitionPair* function in the following four steps.

Step 1. Use the *zip* function to identify each *Tablet* with a number. It simulates the situation that we use the reducer number to select the output data.

Step 2. Get all possible combinations of the elements from *lTable* and *rTable*, just like the Cartesian product of two *Tables*. In fact, we need all possible combinations of *lTablets* and *rTabelts*.

Step 3. Filter the useless combinations with the *partitionSelector* function.

Step 4. Remove the *Num* part from pair (*Tablet, Num*).

The definition of *getPartitionPair* is as in Algorithm 12.

5.5. *moduleMerge*.

We implement the *moduleMerge* function in the following three steps.

Step 1. Get all the combinations for *Tablets* from *lTable* and *rTable* with the *getPartitionPair* function.

Step 2. Merge two *Tablets* from different *Tables* with the *mergeTwoPartition* function. The outermost *map* application corresponds to the parallel merge tasks. The innermost *map* application corresponds to a merge task where the *mergeTwoPartition* function is executed.

Step 3. Concatenate all the partitions that produced by a merger with the *foldl* function.

In the definition of the *moduleMerge* function, we use some concepts about type *Map* and its operations. The function *foldl* has three parameters: a binary function, an initial value, and a *Map* type value. It returns a result that is the same type as the initial value. The *union* function combines two dictionaries into one dictionary. The value *empty* is an empty dictionary. The definition of *moduleMerge* is as in Algorithm 13.

5.6. *iterationLogic and partitionSelector*.

The *iterationLogic* and *partitionSelector* functions are implemented according to different join algorithms as in Algorithm 14.

```
partitionSelcetor :: (Map K21 V31, Int) → (Map K22 V32, Int) → Bool
partitionSelcetor (lTablet, left) (rTablet, right) = True
iterationLogic :: KV1 → KV2 → Bool
iterationLogic (K21, V31)(K22, V32) = True
```

Algorithm 14

```
type K21 = (Int, String)     --Employee key is (emp-id,dept-id)
type V31 = Float             --Employee value is bonus
type K22 = String            --Department key is dept-id
type V32 = Float             --Department value is bonus-adjustment
type K23 = Int               --Bonus key is emp-id
type V33 = Float             --Bonus value is bonus-final
```

Algorithm 15

```
lTablet1 = fromList [((2, "A"), 0), ((3, "A"), 250), ((1, "B"), 150)]     --Tablet1 of Table1
lTablet2 = fromList [((4, "B"), 0), ((7, "C"), 200), ((6, "D"), 100)]     --Tablet2 of Table1
lTablet3= fromList [((5, "A"), 300), ((8, "C"), 150), ((9, "D"), 0)]      -- Tablet3 of Table1
lTable = [lTablet1, lTablet2, lTablet3]                                   --Table1
rTablet1 = fromList [("A", 0.95), ("B", 1.15)]                            --Tablet1 of Table2
rTablet2 = fromList [("C", 1.00), ("D", 1.25)]                            --Tablet2 of Table2
rTable = [rTablet1, rTablet2]                                             --Table2
```

Algorithm 16

The *iterationLogic* function judges whether two *Records* satisfy the merge condition. As we can see in this definition, we assume that it is always true no matter what the input data is. This is correct when we want to implement nested-loop join. The definition will change to $K21{=}K22$ if sort-merge join is implemented.

The *partitionSelector* function selects those *Tablets* that need to be merged. Just like the *iterationLogic* function, the return value is true when we want to implement nested-loop join. We will change the definition to *left==right* when sort-merge join is needed, on condition that two Map-Reduce phases use the same partitioners.

As we can see in this section and last section, our method to describe the Merge module is "firstly design the types and then define the functions." The types are designed from top to down, while the definitions are defined from bottom to up. At last, we get the abstract description of the Merge module by using Haskell as shown in Figure 4.

6. Case Study

In this section, an example of how to use our description is designed to demonstrate the proposed method. Then a brief analysis on two join algorithms that have been implemented in Map-Reduce-Merge is given, including nested-loop join and sort-merge join.

There are two tables in this example: Employee and Department. We use *Table*1 and *Table*2 to represent them, respectively. The primary keys of the tables are shown in bold in Figures 5 and 6. One possible query is to compute the employee final bonus that is the product of bonus in *Table*1 and bonus adjustment in *Table*2. This query result is stored in *Table*3. To accomplish this query, we take steps as follows.

Firstly, we divide the record in a table into two parts, corresponding to *Key* and *Value* in Map-Reduce, respectively, and assign every attribute with a Haskell type. In *Employee*, we chose the composition of *emp-id* and *dept-id* as *Key* and the others as the *Value*. As to *Department*, we chose *dept-id* as *Key* and the others as the *Value*. All the types of *Key* and *Value* emerging in the tables are defined as in Algorithm 15.

Notice that the *Key* we use in *Map-Reduce* is not the same as the key we use in databases. In our example, we use the same *Key* as the primary key in *Table*1, while in *Table*2 we chose not to. In the subsequent steps, we implement the nested-loop join with our description. The way to process sort-merge join is similar except that we use other definitions of *iterationLogic* and *partitionSelector*, which will be discussed later.

Secondly, we construct the input data by modifying the table to the form of list of Maps as follows. *Table*1 consists of three *Tablets*, and *Table*2 consists of two *Tablets* (see Algorithm 16).

Thirdly, we define the functions match and *mergerResult*. In the *match* function, we guarantee that the employee *dept-id* is equal to the department *dept-id*. Only those *Record* pairs that satisfy this condition can be merged. In the *mergeResult* function, we implement the product of bonus and bonus adjustment to get the final bonus. The types and definitions of these two functions are shown as in Algorithm 17.

```
import Data.Map (Map, empty, union, fromList, toList)
moduleMerge :: [Map K21 V31] → [Map K22 V32] → [Map K23 V33]
moduleMerge lTable rTable =
    map (foldl union empty)
    $ map (map (uncurry mergeTwoPartition))
    $ getPartitionPair lTable rTable
    where
        mergeTwoPartition :: Map K21 V31 → Map K22 V32 → Map K23 V33
        mergeTwoPartition lTablet rTablet =
            fromList
            $ map (\(Just x)→x) $ filter (\x→x/=Nothing)
            $ map (uncurry mergeTwoRecord) $ getPair lTablet rTablet
            where
                mergeTwoRecord :: KV1 → KV2 → Maybe KV3
                mergeTwoRecord kv1@(k21, v31) kv2@(k22, v32)
                    | match k21 k22 = Just $ mergeResult kv1 kv2
                    | otherwise = Nothing
                getPair :: Map K21 V31 → Map K22 V32 → [(KV1, KV2)]
                getPair lTablet rTablet =
                    filter (uncurry iterationLogic)
                    [(pairLeft, pairRight) | pairLeft ← toList lTablet,
                    pairRight ← toList rTablet]

        getPartitionPair :: [Map K21 V31] → [Map K22 V32]
                             → [[(Map K21 V31, Map K22 V32)]]
        getPartitionPair lTable rTable =
            map (map (\((lTablet, lNum), (rTablet, rNum))→(lTablet,
rTablet)))
            $ map (filter (uncurry partitionSelcetor))
            $ [[(pairLeft, pairRight) | pairRight ← rTableNum] |
            pairLeft ← lTableNum]
            where
                lTableNum = zip lTable [1, 2..]
                rTableNum = zip rTable [1, 2..]
                partitionSelcetor :: (Map K21 V31, Int)
                                     → (Map K22 V32, Int)
                                     → Bool
                partitionSelcetor (lTablet, left) (rTablet, right) = True
```

Figure 4: The abstract description for *moduleMerge*.

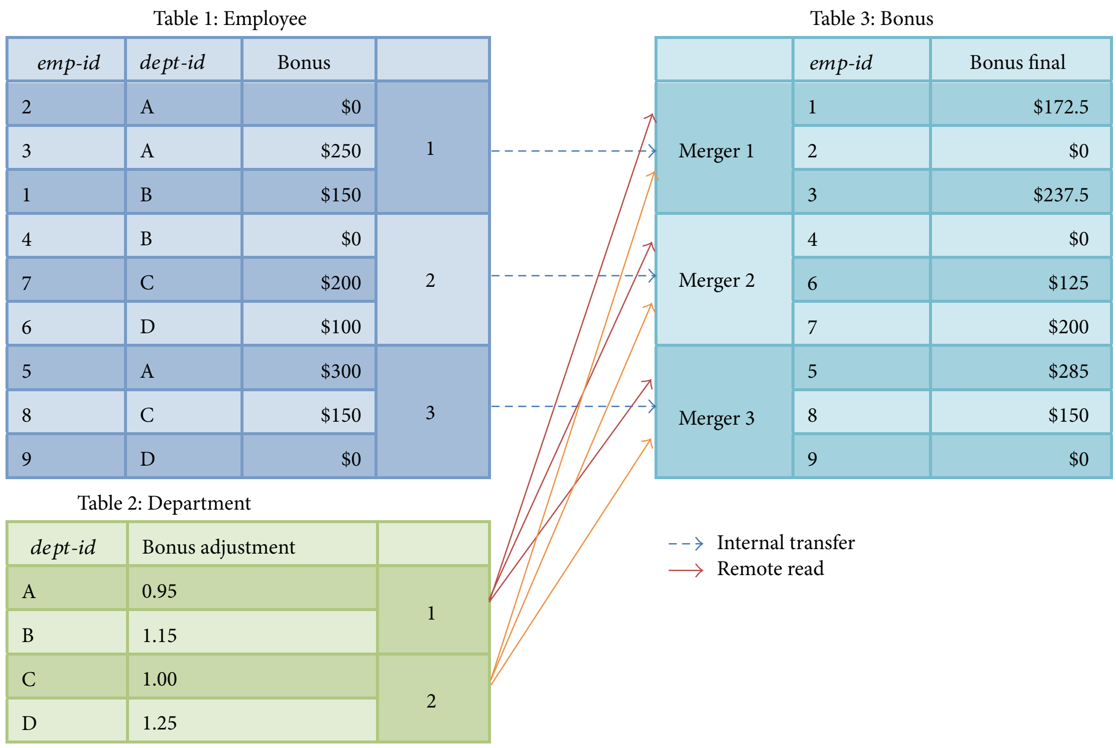

Figure 5: Data flow of nested-loop join.

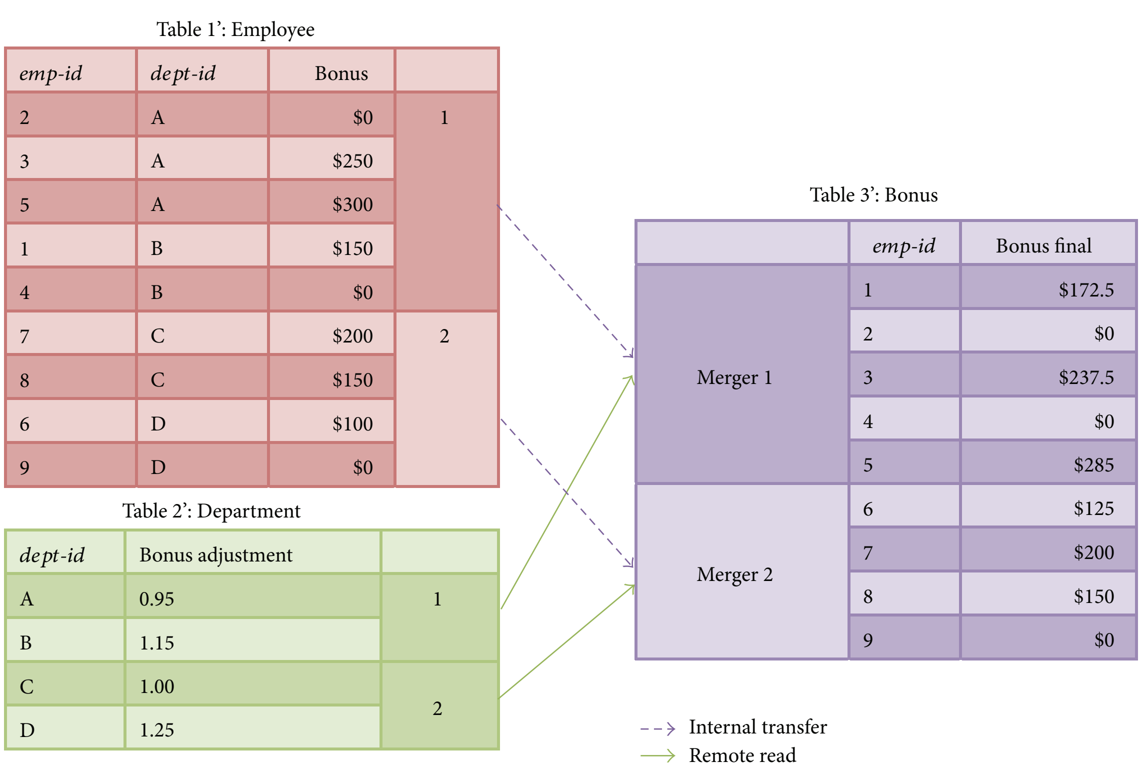

Figure 6: Data flow of sort-merge join.

```
match ::  K21          --Key21 as input data
          → K22        --Key22 as input data
          → Bool       --Satisfy the merge condition or not
match K21 K22 = let (int, str) = K21 in if str == K22 then True else False
mergeResult ::  KV1          --Record1 as input data
                → KV2        --Record2 as input data
                → KV3        --Record3 as output data
mergeResult (K21, V31) (K22, V32)= let (int, str)= K21 in (int, V31*V32)
```

Algorithm 17

Finally, we run the following command in Haskell compiler WinGHCi:

Prelude > moduleMerge lTable rTable.

The result is as follows. It corresponds to Table 3 in Figure 5:

[*fromList*[(1,172.5),(2,0.0),(3,237.5)],*fromList*[(4,0.0),(6,125.0),(7,200.0)],*fromList*[(5,285.0),(8,150.0),(9,0.0)]].

It shows that our description can implement these two join algorithms. The corresponding dataflow graphs are Figures 5 and 6. We set the result of the *partitionSelector* function to be true in order to ensure that every *Tablet* from *Table*1 can be merged with all tables in *Table*2. On the other hand, we set the function body of *partitionSelector* to be *left==right* so as to map reducers and mergers in a one-to-one relationship. We use the same strategy to define the *iterationLogic* function. The main difference between sort-merge join and nested-loop join is that the input of sort-merge join has been sorted, while the input of nested-loop join has not been. In this example, we take advantage of the combining phase strategy in Map-Reduce-Merge framework to reduce the remote read between reducers and mergers. Since the size of *Table*1 is much bigger than the size of *Table*2, we combine the mergers with the reducers into same workers.

As we can see there are two processors and two iterators in the Merge module. Hence, Map-Reduce-Merge can implement two-way join algorithms. If we want to implement multiway join algorithms, a join tree (or a Map-Reduce-Merge workflow) is needed. According to the data flow of Figures 5 and 6, we can find that using sort-merge join can decrease the remote reads than nested-loop join. In Map-Reduce-Merge, sort-merge join is more efficient than nested-loop join when processing equal join. When processing more complicated join, the nested-loop join algorithm is in need.

7. Conclusions

Map-Reduce-Merge is an improved work based on Google's Map-Reduce programming model. It improves the ability to express and to process join operation among multiple heterogeneous datasets. At the same time, it increases the complexity to understand the execution flow of a job. This paper presents a rigorous description of Map-Reduce-Merge to abstract the fundamental functions for Map-Reduce-Merge using Haskell. Based on the abstract description, we analyze the characteristics of Map-Reduce-Merge programming model. On one hand, our work can help with an unambiguous understanding of Map-Reduce-Merge and provide strong theoretical basis for designing more efficient parallel programming model to process join operation. On the other hand, programmers can use our description as a specification in software development. Our result can ensure the correctness and robustness of the software with Haskell strong type checking and type inference.

Since this paper mainly concentrates on describing the dataflow in Map-Reduce-Merge, an important future direction is to introduce some control parameters into our description to improve its flexibility and usability. In addition, cost information can cooperate with our description to estimate the performance of a Map-Reduce-Merge job.

Acknowledgments

This work was supported by the Specialized Research Fund for the Doctoral Program of Higher Education of China under Grant no. 20120061120059; the China Postdoctoral Science Foundation under Grant no. 2011M500612; the Key Program for Science and Technology Development of Jilin Province of China under Grant no. 20130206052GX; the National Natural Science Foundation of China under Grant nos. 61300049, 61070084, and 61100090; the Program for New Century Excellent Talents in University under Grant no. NCET-10-0436; the Science and Technology Development Program of Jilin Province of China under Grant no. 20101509; the Fundamental Research Funds for the Central Universities under Grant nos. 201103124, 201103133; and the "985 Project" of Jilin University of China.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] P. Wang, D. Meng, J. Zhan, and B. Tu, "Review of programming models for data-intensive computing," *Journal of Computer Research and Development*, vol. 47, no. 11, pp. 1993–2002, 2010.

[3] J.-J. Li, J. Cui, D. Wang, L. Yan, and Y.-S. Huang, "Survey of MapReduce parallel programming model," *Chinese Journal of Electronics*, vol. 39, no. 11, pp. 2635–2642, 2011.

[4] Y. Li, Y. Wang, and L. Bao, "FACC: a novel finite automaton based on cloud computing for the multiple longest common subsequences search," *Mathematical Problems in Engineering*, vol. 2012, Article ID 310328, 17 pages, 2012.

[5] X. Wang, Y. Wang, and H. Zhu, "Energy-efficient multi-job scheduling model for cloud computing and its genetic algorithm," *Mathematical Problems in Engineering*, vol. 2012, Article ID 589243, 16 pages, 2012.

[6] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1029–1040, Beijing, China, June 2007.

[7] J. Wang, T. Wang, D. Yang, and H. Li, "A filter-based multi-join algorithm in cloud computing environment," *Journal of Computer Research and Development*, vol. 48, supplement, pp. 245–253, 2011.

[8] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 495–506, Indianapolis, Ind, USA, June 2010.

[9] D. Jiang, A. K. H. Tung, and G. Chen, "MAP-JOIN-REDUCE: toward scalable and efficient data analysis on large clusters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1299–1311, 2011.

[10] A. Okcan and M. Riedewald, "Processing theta-joins using MapReduce," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 949–960, Athens, Greece, June 2011.

[11] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MaPreduce," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 975–986, Indianapolis, Ind, USA, June 2010.

[12] X.-P. Qin, H.-J. Wang, X.-Y. Du, and S. Wang, "Big data analysis—competition and symbiosis of RDBMS and MapReduce," *Journal of Software*, vol. 23, no. 1, pp. 32–45, 2012.

[13] S. Wang, H.-J. Wang, X.-P. Qin, and X. Zhou, "Architecting big data: challenges, studies and forecasts," *Chinese Journal of Computers*, vol. 34, no. 10, pp. 1741–1752, 2011.

[14] S.-L. Gui, L. Luo, Y. Li, M. Yu, and J.-H. Xu, "Schedulability analysis tool for distributed real-time systems based on automata theory," *Journal of Software*, vol. 22, no. 6, pp. 1236–1251, 2011.

[15] J. Wang, M. Yin, and W. Gu, "Fuzzy multiset finite automata and their languages," *Soft Computing*, vol. 17, no. 3, pp. 381–390, 2013.

[16] X. Liu, L. Yang, M.-X. Pan, and L.-Z. Wang, "Scenario-driven service behavior manipulation," *Journal of Software*, vol. 22, no. 6, pp. 1185–1198, 2011.

[17] W. Gu, G. Li, and M. Yin, "Extending fuzzy soft sets with fuzzy description logics," *ICIC Express Letters, Part B*, vol. 2, no. 5, pp. 1001–1007, 2011.

[18] R. Lämmel, "Google's MapReduce programming model—revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, 2008.

[19] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: parallel analysis with Sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[20] F. Palmieri, L. Buonanno, S. Venticinque, R. Aversa, and B. Di Martino, "A distributed scheduling framework based on selfish autonomous agents for federated cloud environments," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1461–1472, 2013.

[21] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Scheduling strategies for optimal service deployment across multiple clouds," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1431–1441, 2013.

[22] D. Cai and M. Yin, "On the utility of landmarks in SAT based planning," *Knowledge-Based Systems*, vol. 36, pp. 146–154, 2012.

[23] D. Cai, J. Sun, and M. Yin, "Conformant planning heuristics based on plan reuse in belief states," in *Proceedings of the 23rd AAAI Conference on Artificial Intelligence and the 20th Innovative Applications of Artificial Intelligence Conference*, pp. 1780–1781, Chicago, Ill, USA, July 2008.

[24] C. Zhang and J. Sun, "An alternate two phases particle swarm optimization algorithm for flow shop scheduling problem," *Expert Systems with Applications*, vol. 36, no. 3, pp. 5162–5167, 2009.

[25] C. Zhang, J. Ning, and D. Ouyang, "A hybrid alternate two phases particle swarm optimization algorithm for flow shop scheduling problem," *Computers and Industrial Engineering*, vol. 58, no. 1, pp. 1–11, 2010.

[26] B. O'Sullivan, J. Goerzen, and E. Don Stewart, *Real World Haskell*, O'Reilly Media, 2008.

[27] F. Chang, J. Dean, S. Ghemawat et al., "Bigtable: a distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, article 4, 2008.